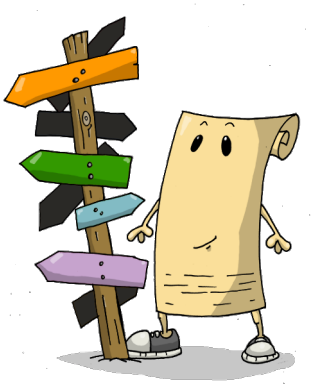
User Manual

for

Textweiser SDK

A software to classify text

Covers version 1.3.0



Lingua Systems
NATURAL LANGUAGE PROCESSING

Textweiser SDK User Manual, published April 23, 2014.

Copyright © 2010-2014, Lingua-Systems Software GmbH

Lingua-Systems Software GmbH, Gerichtsstraße 42, 44649 Herne, Germany, *info@lingua-systems.com*

All rights reserved, especially changing or publishing parts of this manual needs prior written permission of the copyright owner.

The rights to reproduce and publish unchanged copies in any form, to translate or to present the manual are granted.

Mentioned hard- and software as well as companies may be trademarks of their respective owners. Use of a term in this manual should not be regarded as affecting the validity of any trademark or service mark. A missing annotation of the trademark may not lead to the assumption that no trademark is claimed and may thus be used freely.

Great effort has been made in writing this manual. However, faults cannot be excluded in general. For any loss or damages caused or alleged to be caused directly or indirectly by errors or omissions in this manual, the authors and the publisher assume no responsibility and cannot be held liable. Neither can the authors or the publisher be held liable for the content or changes of content concerning the linked websites. The links have been carefully chosen and proved at the preparation of the manual.

If you have problems using the links or get aware of any faults, feel free to give a brief hint on it via *support@lingua-systems.com*.

Contents

1. Introduction **6**

2. Installation **7**
2.1. Requirements . . . 7
2.2. What Will Be Installed . . . 7
2.3. Installing the Software . . . 7
2.4. Deinstalling the Software . . . 7

3. Hints on the Usage of Textweiser **8**
3.1. Working with Category Structures . . . 8
3.1.1. Flat Category Structures . . . 8
3.1.2. Mono-hierarchical Category Structures / Taxonomies . . . 8
3.2. Types of Probability Calculation . . . 9
3.3. Common Workflow . . . 10
3.4. Encryption of the Database Connection . . . 10
3.4.1. Microsoft SQL Server . . . 11

4. Application Programming Interface **12**
4.1. Overview . . . 12
4.1.1. Functions for Administration . . . 12
4.1.2. Functions for Resource Handling . . . 13
4.1.3. Functions for Learning . . . 13
4.1.4. Functions for Classification . . . 14
4.1.5. Auxiliary Functions . . . 14
4.2. Configuration File . . . 15
4.3. Important Data Structures . . . 16
4.3.1. Textweiser Object tw_t . . . 16
4.3.2. Classification Result tw_prob_t . . . 16
4.3.3. Classification Options tw_classify_opt_t . . . 16
4.3.4. Configuration Data tw_config_t . . . 17
4.4. Function Reference . . . 19
4.4.1. tw_add_category() and tw_delete_category() . . . 19
4.4.2. tw_backup_db() and tw_restore_db() . . . 19
4.4.3. tw_classify(), tw_classify_v2(), tw_classify_file() and tw_classify_file_v2() . . . 20
4.4.4. tw_create_db() and tw_erase_db() . . . 20
4.4.5. tw_free() . . . 21
4.4.6. tw_free_categories() . . . 21
4.4.7. tw_free_config_t() . . . 21
4.4.8. tw_free_prob_t() . . . 21
4.4.9. tw_get_categories() . . . 22
4.4.10. tw_init() . . . 22
4.4.11. tw_learn() and tw_learn_file() . . . 22
4.4.12. tw_optimize_db() . . . 23
4.4.13. tw_parse_config() . . . 23
4.4.14. tw_rename_category() . . . 23
4.4.15. tw_strerror() . . . 24
4.4.16. tw_unlearn() and tw_unlearn_file() . . . 24
4.4.17. tw_version() . . . 25
4.4.18. tw_version_string() . . . 25
4.5. Error Handling . . . 26
4.5.1. tw_errno_t Named Error Constants . . . 27

4.6. Hints on Application Development 28
4.6.1. Determining Textweiser's Version 28

5. Commandline Interface **29**
5.1. Connecting to the Database 29
5.2. Common Options 30
5.3. tw-admin: Textweiser Administration 30
5.3.1. Usage Example 31
5.4. tw-learn: Learn Category Characteristics 31
5.4.1. Usage Example 32
5.5. tw-classify: Classify Unknown Documents 33
5.5.1. Usage Example 34
5.6. tw-backup: Backup and Restore the Database 35
5.6.1. Usage Example 35

A. Example Application: add-learn.c **36**

B. Example Application: classify.c **38**

C. References **40**

About this Manual

This manual addresses users with experience in C/C++ programming and at least a basic knowledge of library usage as well as users who use the commandline applications.

The manual provides a short introduction to the library and the applications, followed by instructions how to install the Textweiser software package. Afterwards some hints on the usage of a text classifier are given, before the complete interface (API) is introduced along with the possibilities of error handling. Finally, the commandline applications are introduced including usage examples.

For a quickstart have a look at the documentation of the application programming interface (chapter 4 on page 12).

Administrators who want to install the software can obtain all necessary information from chapter 2, page 7.

1. Introduction

Textweiser is a text classifier that assigns unknown text documents to categories.

The software's administrator prepares the software for usage first: all categories have to be added to the system. After adding the categories they must each be trained with a set of representative documents – at least ten for each category. Textweiser analyses the documents and extracts the relevant information needed to classify unknown documents afterwards. The information is stored in a database.

When training is accomplished, Textweiser can classify unknown documents and the system is ready for use. Textweiser can for example be used to suggest categories, automatically route emails or more generally in the field of document management.

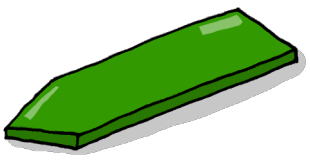
The software can handle both flat and mono-hierarchical category structures ("taxonomies"). The handling of hierarchies is fully supported (see chapter 3.1 on page 8).

An intuitive interface to the library allows you to integrate Textweiser easily. The C/C++ library is thread-safe and provides access to all functions needed to make use of a text classifier within your own application.

Additionally Textweiser comes along with a set of commandline applications. These applications allow users to classify text and administrators to maintain the system. The applications may be used automated within scripts as well.

Any input passed to Textweiser has to be plain text and should be encoded in UTF-8.

The text data is preprocessed language dependent to optimize the results. Therefore it is recommended to use Textweiser only with supported language data. Other languages can be processed as well, but results are likely to be less precise. A list of supported languages is provided in the software specification.



2. Installation

2.1. Requirements

Textweiser requires the system's standard C and thread libraries. Additional requirements depend on the database used.

With *SQLite*, no further dependencies occur, as this database software is already included.

With *Microsoft SQL Server*, Textweiser additionally requires both the standard ODBC (`odbc32.dll`) and the *SQL Server Native Client* (10.0) library to be installed. Hints on installation are available on the *Microsoft Developer Network (MSDN)* (see appendix C on page 40).

2.2. What Will Be Installed

The Textweiser SDK contains a dynamic library (DLL/SO), its header file, the code of an example application and this manual.

The Software Development Kit for Linux contains the following files:

```
./bin:
tw-admin*  tw-backup*  tw-classify*  tw-learn*

./doc:
examples/  LICENSE  manual-sdk-eng.pdf

./doc/examples:
example_add-learn.c  example.cfg  example_classify.c
example_get_categories.c  example_init.c  example_parse_config.c

./include:
tw.h

./lib:
libtw.so@  libtw.so.1@  libtw.so.1.1.0
```

2.3. Installing the Software

Textweiser SDK is provided as a compressed archive, either in "Zip" or "tar.gz" form, depending on the target platform.

To install the software, just unpack the archive to a directory of your choice and add the library and header files to your project.

2.4. Deinstalling the Software

To deinstall the software, just remove the directory you unpacked Textweiser SDK to.

3. Hints on the Usage of Textweiser

Before putting a text classifier into operation it is necessary to plan the deployment first. If planning of the resulting structure of categories is accomplished, you can start preparing the text classifier. Nevertheless it is possible to change the structure during operation. Textweiser allows to add new categories or rename and delete existing ones.

One of the most important factors for accuracy of the classification results is the training of the classifier. During training the system learns the characteristics of representative documents for each category. It is recommended to choose at least ten documents each.

When training is complete, the software can be used to classify unknown documents. Textweiser provides a list of categories a document may belong to along with their probabilities. The number of results can be defined by the application that uses the Textweiser library. This way the library may be used to classify a document automatically when choosing one result or provide a list of suggestions to the end user.

3.1. Working with Category Structures

Textweiser supports both flat and mono-hierarchical category structures ("taxonomies").

3.1.1. Flat Category Structures

Flat category structures cannot express any hierarchical relations. All categories are located on the same level, as the following diagram shows.

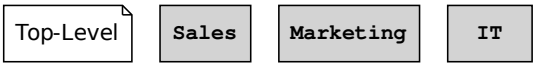


Figure 1: Example of a flat Category Structure

A flat structure is easy to plan and implement. It is suitable for systems that have a small amount or medium of categories.

3.1.2. Mono-hierarchical Category Structures / Taxonomies

Relations between categories can be expressed using mono-hierarchical structures ("taxonomies"). The relations result in a tree structure with a set of top-level and sub-level categories. Each sub-level category may only have one top-level category but may itself have several sub-level categories.

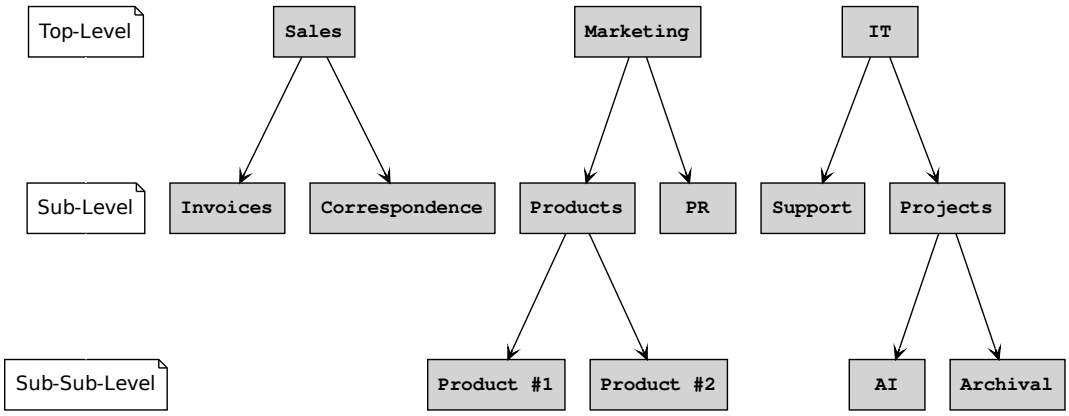


Figure 2: Example of a mono-hierarchical Structure ("Taxonomy")

To handle mono-hierarchical structures, Textweiser provides an explicite notation for hierarchical relations. The categories are separated by "::".

For example, the category "Archival" with its top-level categories "Projects" and "IT" is addressed with "*IT::Projects::Archival*".

When hierarchies are used with this notation, Textweiser automatically organizes the data accordingly:

→ Add a category
When a sub-level category is added to the system, any top-level categories are added as well if they have not existed yet.

→ Learn a document
When learning a document for a sub-level category, the data is assigned to all affected top-level categories as well. A document learned for "*IT::Projects::Archival*" is also assigned to "*IT::Projects*" und "*IT*".

→ Rename a category
If a top-level category is renamed, all existing sub-level categories are renamed accordingly, so the relations between the documents stay the same.

→ Delete a category
Deleting a top-level category deletes all its sub-categories as well.

3.2. Types of Probability Calculation

Textweiser provides two types of probability calculation for classification results: absolute or distributed.

The *absolute calculation* determines the probability an input document belongs to a category – *independent* from all other categories within the set of classification results.

The *distributed calculation* determines the probability taking every other category and their *relation* within the set of results into account. The distributed probability sums up to the absolute probability of the most probable category, which not necessarily needs to be 100%.

#	Category	Absolute	Distributed
1	Technology	100.00%	32.14%
2	Economy	69.66%	22.39%
3	Economy::Markets	61.61%	19.80%
4	Sport	44.69%	14.36%
5	Sport::Football	35.17%	11.30%

Figure 3: Example of different probability calculations

3.3. Common Workflow

A typical workflow includes the following steps:

1. Create a database
2. Add categories
3. Learn documents
4. Optimize database
5. Classify

Optimizing the database increases performance and accuracy. It is recommended to do an optimization whenever you learned a set of documents, deleted a category or unlearned a document.

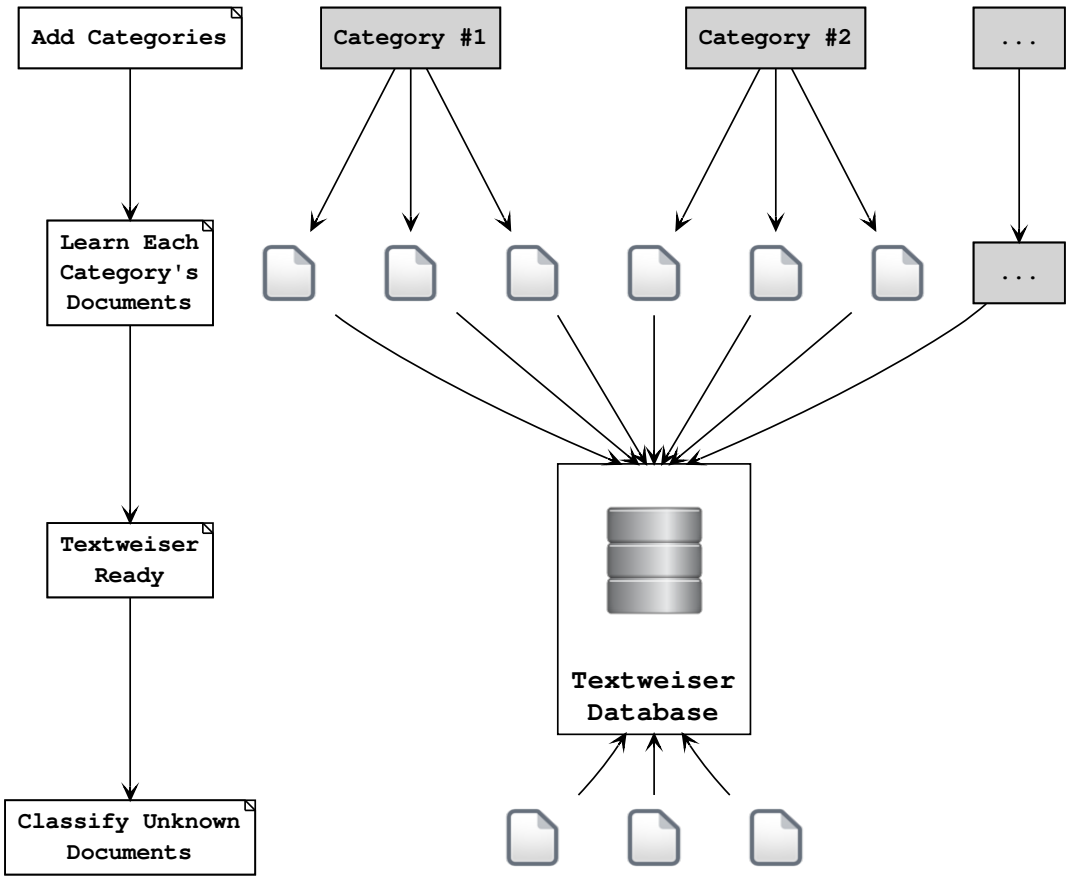


Figure 4: Common Workflow

Additionally functions are provided to maintain the database:

→ Generate a backup

→ Restore data from a backup

→ Delete data from a database

3.4. Encryption of the Database Connection

Depending on the database used, Textweiser may provide the option to encrypt the connection to the database. This way, data will be transmitted over the network securely.

Textweiser supports encryption if one of the following databases is used:

1. Microsoft SQL Server

   Textweiser does not implement encryption itself but relies on the used database driver for this task. However, Textweiser will assure that the driver is configured to use solely encrypted connections on request.

   Further information on how to configure Textweiser to use encrypted connections can be found in chapter 4.3.4 on page 17 and – concerning the commandline applications – in chapter 5.1 on page 29.

3.4.1. Microsoft SQL Server

Microsoft SQL Server provides SSL secured connections using certificates. By default, a certificate provided by the database server is validated and the connection will be rejected if the certificate fails to validate and Textweiser has been configured to use encryption.

Whenever no certificate has been assigned to the server, Microsoft SQL Server will generate a self-signed certificate that may be used for encryption (without validation). If you intend to use this certificate, Textweiser has to be configured to instruct the database driver to trust the server certificate *without* validation.

To accomplish this, the member `encrypt` of the used variable of type `tw_config_t` has to be set to both `TW_ENCRYPT_ON` and `TW_ENCRYPT_TRUST_CERT`. When using the commandline applications, use both the `--encrypt` and `--trust-cert` options. In a Textweiser configuration file, setting the key "`encrypt`" to a value of "`on, trust-cert`" is sufficient to allow encryption using self-signed certificates.
The variables and options will be described in later chapters of this manual again on pages 15 (configuration file), 17 (variables und flags) and 29 (options of the commandline applications).

For further information on the configuration of the database server, please refer to your Microsoft SQL Server documentation and the links relevant articles on MSDN as referred to in appendix C on page 40.

4. Application Programming Interface

4.1. Overview

Textweiser SDK's C/C++ library provides an API that is intuitive to use and allows integration into applications easily. All functions and data structures are prefixed *tw_* to avoid confusions and collisions with other third party library functions and are defined in the header file *tw.h*.

Input passed to the library is expected to be plain text and encoded in UTF-8. It is recommended to use Textweiser only with supported languages (see the software specification). Other languages can be processed nevertheless, but important tasks of linguistic preprocessing will be missing, so getting less accurate results is likely.

The functions can be divided into five categories: administration, resource handling, learning, classification and auxiliaries.

4.1.1. Functions for Administration

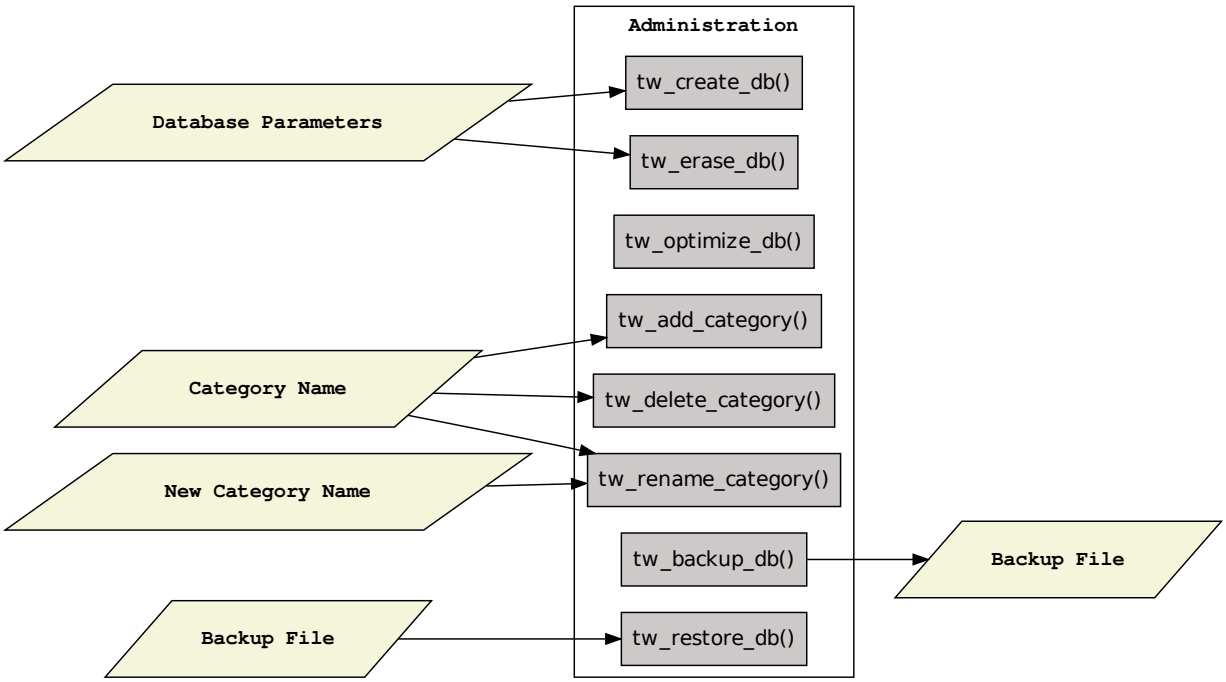


Figure 5: Flowchart of the Functions for Administration

New Textweiser databases can be created with *tw_create_db()*. When training of categories is completed, the database can be optimized in performance and accuracy using *tw_optimize_db()*. Categories can be added, renamed and deleted using tw_add_category(), *tw_delete_category()* or *tw_rename_category()*.

Additional functions to maintain the database are *tw_backup_db()* and *tw_restore_db()* which create backups of the Textweiser database and restore the data if necessary. *tw_erase_db()* deletes any Textweiser data from a database.

4.1.2. Functions for Resource Handling

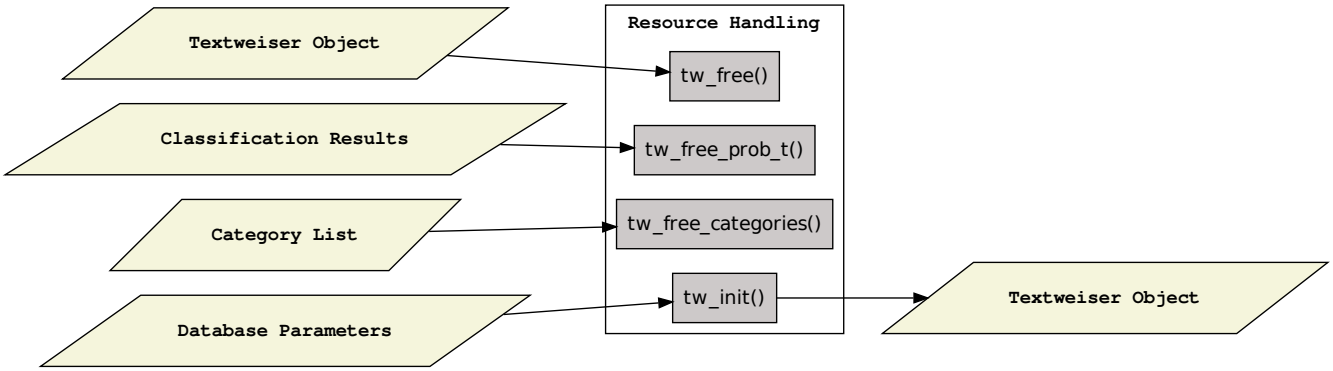


Figure 6: Flowchart of the Functions for Resource Handling

tw_init() initializes a new Textweiser object and opens a connection to the database. The object and its allocated memory can be freed with *tw_free()* if it is no longer needed – this function closes the database connection as well.

The allocated memory used for classification results stored in `tw_prob_t` can be freed with *tw_free_prob_t()*. Accordingly, *tw_free_config_t()* frees memory used by a `tw_config_t` data structure and *tw_free_categories()* frees the memory used by a category listing.

4.1.3. Functions for Learning

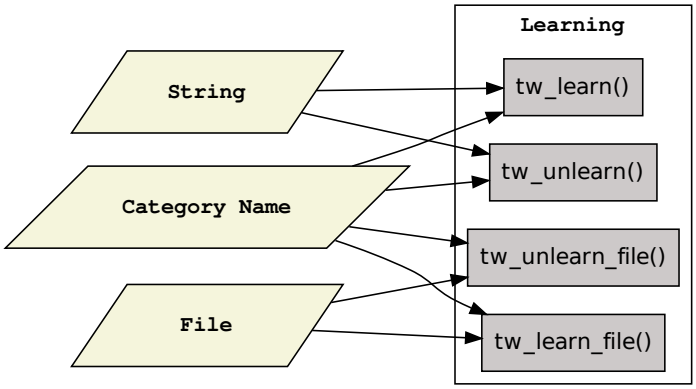


Figure 7: Flowchart of the Functions for Learning

In order to assign unknown documents to a category, Textweiser has to learn the characteristics of each category with the help of representative documents. Use *tw_learn()* or *tw_learn_file()* to train Textweiser.

If a document was learned by mistake, use *tw_unlearn()* or *tw_unlearn_file()* to undo the training.

4.1.4. Functions for Classification

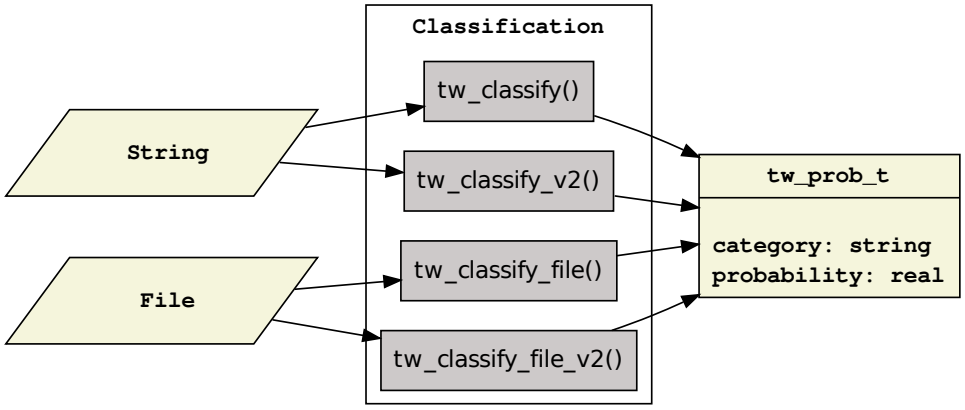


Figure 8: Flowchart of the Functions for Classification

As soon as all categories have been trained, Textweiser can classify documents. The functions *tw_classify()*, *tw_classify_v2()*, *tw_classify_file()* and *tw_classify_file_v2()* assign unknown text to categories.

4.1.5. Auxiliary Functions

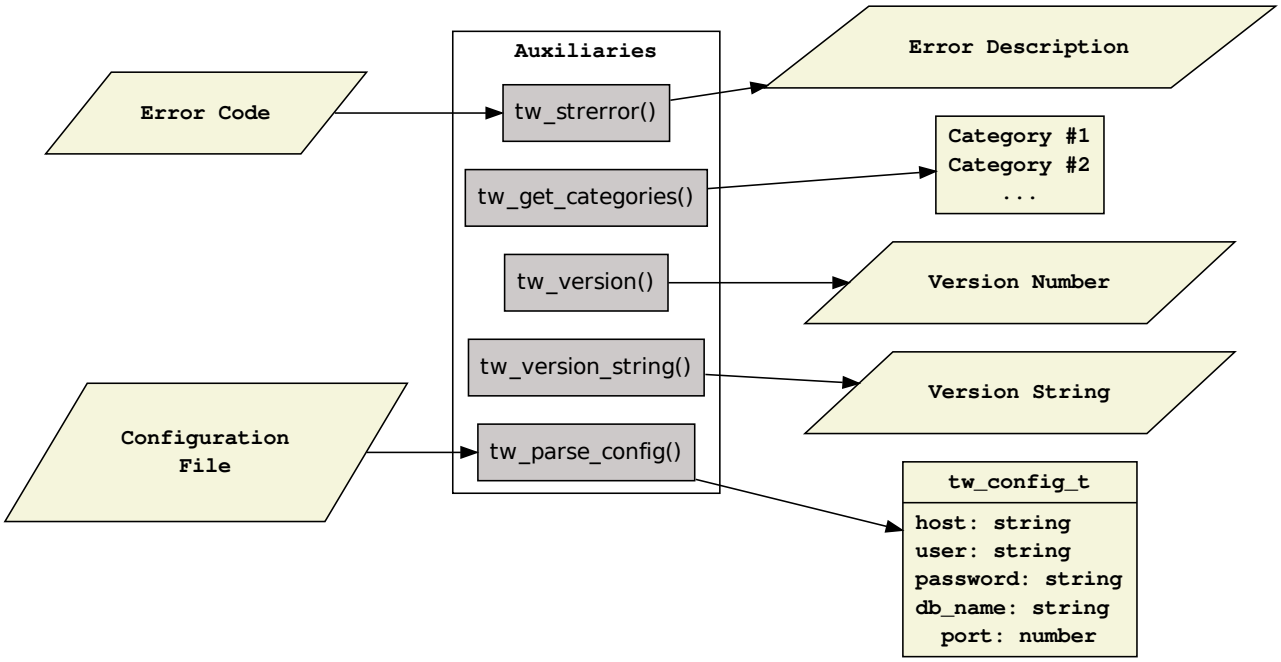


Figure 9: Flowchart of the Auxiliary Functions

tw_strerror() provides an English error message for error codes used by Textweiser.

A list of all categories can be obtained with *tw_get_categories()*.

tw_version() and *tw_version_string()* provide the library's version at runtime.

tw_parse_config() reads and evaluates database parameters stored in a configuration file.

4.2. Configuration File

To ease managing the connection to a database, all parameters necessary can be stored in a configuration file.

Both the function for parsing and the data structure are described in later chapters. This chapter describes the syntax of the configuration file only.

The configuration file contains simple key/value pairs for all parameters:

host	Hostname of the database server
user	Username for database authentification
passwd	Password for database authentification
db_name	Name of the Textweiser database
port	Port number of the database server
instance	Name of the Microsoft SQL Server instance
encrypt	Configuration whether and how encryption should be used

Each value is associated to a key by assignment (equal sign) and can optionally be written in single or double quotes. Empty lines and whitespace at the start or end of a line are ignored. Lines starting with (#) are interpreted as comments.

Special attention has to be paid to the key `encrypt`, which may only be set to one of the following predefined values:

Values	Description	Comment
"off", "no"	Disable encryption	Default
"on", "yes"	Enable encryption	-
"trust-cert"	Trust certificate	In addition to "on" or "yes"

Figure 10: Valid Values for `encrypt`

The value `trust-cert` has to follow either "on" or "yes" and may be separated by a comma and/or whitespace. For example: "on, trust-cert".

For further information on encryption, please refer to chapter 3.4 on page 10.

```
# Example configuration file (Microsoft SQL Server)

host    = "dbsrv.local"

user    = 'test'
passwd  = 'secret'

db_name = Textweiser

encrypt = on, trust-cert

instance = "SQLEXPRESS"

# port not set -> use default
```

4.3. Important Data Structures

The data structure `tw_errno_t` is described in a separate chapter on error handling (chapter 4.5, page 26).

4.3.1. Textweiser Object tw_t

The data structure `tw_t` contains data that is exclusively used by Textweiser internally. No application should evaluate or change the data directly.

First, you should assign the macro `TW_INITIALIZER` to any variable of type `tw_t` on declaration in order to initialize it with its default values.

The function *tw_init()* then initializes a `tw_t` object for use within the operating environment and connects to the database. A `tw_t` object is expected as an argument by almost every Textweiser function. Use *tw_free()* to free the memory allocated by this object and disconnect from the database.

4.3.2. Classification Result tw_prob_t

The classification results of the functions *tw_classify()*, *tw_classify_v2()*, *tw_classify_file()* and *tw_classify_file_v2()* are given as an array of (pointers to) a `tw_prob_t` data structure and stored to a user-definable memory location. The end of the array is marked with a `NULL` element.

Each `tw_prob_t` data structure contains the name of a category and the probability the document belongs to this category. The elements of the array are sorted descending by probability.

The formal definition of the data structure is as follows:

```
typedef struct
{
    char *category;
    float probability;
} tw_prob_t;
```

4.3.3. Classification Options tw_classify_opt_t

The functions *tw_classify_v2()* and *tw_classify_file_v2()* allow to pass options that influence classification.

Both functions equal their non-"_v2" counterparts if `TW_CLASSIFY_DEFAULT` is passed as an option.

The data structure comprises the following constants: [1]

[1] Mutually exclusive options are presented in the same color.

Constant	Description	Default?
TW_CLASSIFY_DEFAULT	Use defaults	–
TW_CLASSIFY_PABS	Use absolute probabilities	yes
TW_CLASSIFY_PDIST	Use distributed probabilities	no
TW_CLASSIFY_SPARF	On equal probability, sort parent categories first	yes
TW_CLASSIFY_SSUBF	On equal probability, sort subcategories first	no

Figure 11: Constants of the `tw_classify_opt_t` Data Structure

For a detailed explanation of the provided probability calculations, have a look at chapter 3.2 on page 9.

Whenever a set of classification results shares equal probabilities, the options `TW_CLASSIFY_SPARF` and `TW_CLASSIFY_SSUBF` determine the order of the results data structure. `TW_CLASSIFY_SPARF` sorts any parent category in front of its subcategories while `TW_CLASSIFY_SSUBF` provides the opposite sorting: subcategories preceding their parent categories.

The formal definition of the data structure is as follows:

```
typedef enum
{
    TW_CLASSIFY_DEFAULT = 0,
    TW_CLASSIFY_PABS    = 1,
    TW_CLASSIFY_PDIST   = 2,
    TW_CLASSIFY_SPARF   = 4,
    TW_CLASSIFY_SSUBF   = 8
} tw_classify_opt_t;
```

4.3.4. Configuration Data tw_config_t

Any variable of type `tw_config_t` should be initialized on declaration using the macro `TW_CONFIG_INITIALIZER`.

A configuration file can be used to provide all database settings. The data structure `tw_config_t` is used by the function *tw_parse_config()* to store all settings parsed from the configuration file and make them accessible to the application.

The database settings can also be assigned manually. Examples for the assignment of the values can be found in the example applications in appendix A and B.

Whenever settings have been assigned manually, *tw_free_config_t()* must not be used.

The data structure `tw_config_t` allows to store the following settings: name of the database server (`host`), user name (`user`), password (`passwd`), name of the database to use (`db_name`) and the database's port.

If Microsoft SQL Server is used as a database, the instance name of the server can be set using `instance`.

Encrypted connections to the database can be configured by setting `encrypt` to an appropriate value. Textweiser provides three supported, predefined values for this purpose.

Value	Description	Comment
TW_ENCRYPT_OFF	Disable encryption	Default
TW_ENCRYPT_ON	Enable encryption	-
TW_ENCRYPT_TRUST_CERT	Trust certificate	Requires TW_ENCRYPT_ON

Figure 12: Valid Values for `encrypt`

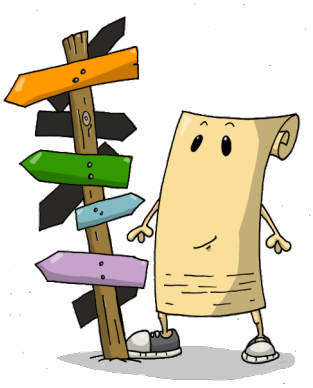
In order to establish an encrypted connection to the database *without* certificate validation, for example to use a self-signed certificate, `encrypt` has to be set to the value that results in either the addition or bitwise OR of `TW_ENCRYPT_ON` and `TW_ENCRYPT_TRUST_CERT`. For further information, refer to chapter 3.4 on page 10.

The formal definition of the data structure is:

```
typedef struct
{
    char          *host;
    char          *user;
    char          *passwd;
    char          *db_name;
    unsigned int   port;
    char          *instance;
    unsigned char  encrypt;
} tw_config_t;
```

The database name `db_name` has to be encoded in UTF-8.

If the database is SQLite, the parameter `db_name` denotes the path to the database – the path does not necessarily need to be encoded in UTF-8. All other parameters are ignored and should be set to `NULL` and `0` for `port`.



4.4. Function Reference

All of Textweiser's functions and data structures are defined in the header file *tw.h*. The header has to be included in all applications that make use of the following functions.

Two example applications for Textweiser's main functions are included in this manual (see appendix A and B on pages 36 and 38) and in the software distribution.

4.4.1. tw_add_category() and tw_delete_category()

```
tw_errno_t tw_add_category(tw_t *tw, const char *name);

tw_errno_t tw_delete_category(tw_t *tw, const char *name);
```

tw_add_category() adds a new category to a Textweiser database, *tw_delete_category()* deletes an existing category and all its data.

Both functions take a pointer to an initialized Textweiser object (`tw_t`) as a first argument (see chapter 4.3.1 on page 16). The second argument is the name of the category to add or delete.

The category's name (`name`) has to be encoded in UTF-8 and must not exceed a length of 255 bytes.

For hints on using mono-hierarchical category structures please refer to chapter 3.1.2 on page 8.

The functions return an error code that indicates whether the respective function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

Both functions are thread-safe and can thus be used by more than one thread at a time.

Deleting a category cannot be reverted. After deleting a category, *tw_optimize_db()* should be used to update the database.

4.4.2. tw_backup_db() and tw_restore_db()

```
tw_errno_t tw_backup_db(tw_t *tw, const char *out_path);

tw_errno_t tw_restore_db(tw_t *tw, const char *in_path);
```

tw_backup_db() generates a backup of a Textweiser database and stores it to a file, *tw_restore_db()* restores a database from such a backup file.

Both functions take a pointer to an initialized Textweiser object (`tw_t`) as a first argument (see chapter 4.3.1 on page 16). As a second argument *tw_backup_db()* expects a path to a file the backup should be stored in. *tw_restore_db()* expects a path to a previously created backup file.

When *tw_restore_db()* is used, any existing data in the database will be replaced by the data of the backup file.

The functions return an error code that indicates whether the respective function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

Both functions are thread-safe and thus can be used by more than one thread at a time.

tw_backup_db() overwrites a backup file if it already exists.

4.4.3. tw_classify(), tw_classify_v2(), tw_classify_file() and tw_classify_file_v2()

```
tw_errno_t tw_classify(tw_t *tw, const char *str, short n,
                       tw_prob_t ***probs);
tw_errno_t tw_classify_v2(tw_t *tw, const char *str, short n,
                       tw_classify_opt_t opt, tw_prob_t ***probs);

tw_errno_t tw_classify_file(tw_t *tw, const char *path, short n,
                            tw_prob_t ***probs);
tw_errno_t tw_classify_file_v2(tw_t *tw, const char *path, short n,
                            tw_classify_opt_t opt,
                            tw_prob_t ***probs);
```

The functions analyse an input document and calculate the probability how likely a document belongs to a category. A list of categories, sorted descending by probability, is stored to `probs` (see page 16). The user may define the maximum number of results (`n`).

Both functions take a pointer to an initialized Textweiser object (`tw_t`) as a first argument (see chapter 4.3.1 on page 16). The second argument is the text to classify either as a string (*tw_classify()*, *tw_classify_v2()*) or as a file, addressed with the path within the file system (*tw_classify_file()*, *tw_classify_file_v2()*). The parameter `n` defines the maximum number of results to store to `probs`.

The functions *tw_classify_v2()* and *tw_classify_file_v2()* allow to specify additional options using the parameter `opt` (see 4.3.3 on page 16).

The end of the array that contains the results is marked with a `NULL` element. Whenever no results could be determined, `probs` is set to `NULL`. The data structure `tw_prob_t` is described in chapter 4.3.2 on page 16.

The text to classify should be encoded in UTF-8.

The functions return an error code that indicates whether the respective function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

Both functions are thread-safe and thus can be used by more than one thread at a time.

4.4.4. tw_create_db() and tw_erase_db()

```
tw_errno_t tw_create_db(const tw_config_t *cfg);
tw_errno_t tw_erase_db(const tw_config_t *cfg);
```

The function *tw_create_db()* creates a new Textweiser database and initializes it with all necessary structures, *tw_erase_db()* deletes all data from a Textweiser database.

The functions expect a pointer to a `tw_config_t` data structure that contains all settings that are necessary to connect to the database. Detail on `tw_config_t` are given in chapter 4.3.4 on page 17.

The functions return an error code that indicates whether the respective function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

Both functions are thread-safe and thus can be used by more than one thread at a time.

tw_erase_db() does not remove the file an SQLite database is stored in.

4.4.5. tw_free()

```
void tw_free(tw_t *tw);
```

This function closes an open connection to a database and frees all resources used by a Textweiser object.

The function takes a pointer to an initialized Textweiser object (`tw_t`) as an argument (see chapter 4.3.1 on page 16).

The function is thread-safe and thus can be used by more than one thread at a time.

This function has to be used on any supported operating system to free allocated memory and close the database connection.

4.4.6. tw_free_categories()

```
void tw_free_categories(char **cats);
```

This function frees the memory used by a category list, pointed to by `cats`, that was generated by *tw_get_categories()*.

The function is thread-safe and thus can be used by more than one thread at a time.

On *Windows* this function is obligatory to free the allocated memory.

4.4.7. tw_free_config_t()

```
void tw_free_config_t(tw_config_t *config);
```

This function frees the memory allocated by a `tw_config_t` data structure, that has been generated by *tw_parse_config()*.

It expects a pointer to a `tw_config_t` data structure.

The function is thread-safe and thus can be used by more than one thread at a time.

This function must not be used if the `tw_config_t` data structure has been initialized or modified manually.

On *Windows* this function is obligatory to free the allocated memory.

4.4.8. tw_free_prob_t()

```
void tw_free_prob_t(tw_prob_t **probs);
```

This function frees the memory allocated by a list of `tw_prob_t` data structures.

It expects a pointer to a list of `tw_prob_t` data structures as generated by *tw_classify()* and *tw_classify_file()*.

The function is thread-safe and thus can be used by more than one thread at a time.

On *Windows* this function is obligatory to free the allocated memory.

4.4.9. tw_get_categories()

```
tw_errno_t tw_get_categories(tw_t *tw, char ***list);
```

This function generates an array of all categories in a Textweiser database and stores it to `list`.

The function takes a pointer to an initialized Textweiser object (`tw_t`) as a first argument (see chapter 4.3.1 on page 16). The second argument is a memory location the generated array should be stored to.

The end of the generated array is marked with a `NULL` element.

The function returns an error code that indicates whether the function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

The function is thread-safe and thus can be used by more than one thread at a time.

If an error occurs or no categories could be found within the database, the value pointed to by `list` is set to `NULL`.

4.4.10. tw_init()

```
tw_errno_t tw_init(tw_t *tw, const tw_config_t *cfg);
```

This function connects to an existing Textweiser database and initializes a new Textweiser object.

The first argument is a pointer to an uninitialized Textweiser object `tw_t` (see chapter 4.3.1 on page 16). The object is initialized by this function so it is ready for use afterwards.

As a second parameter the function expects a pointer to a `tw_config_t` data structure that contains all settings that are necessary to connect to the database. Detail on `tw_config_t` are given in chapter 4.3.4 on page 17.

You should assign the macro `TW_INITIALIZER` to any variable of type `tw_t` on declaration in order to initialize it with its default values before passing it to *tw_init()* along with the settings of the operating environment.

The function returns an error code that indicates whether the function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

When the Textweiser object is not needed any longer, its memory should be freed with *tw_free()* (see chapter 4.4.5 on page 21).

The function is thread-safe and thus can be used by more than one thread at a time.

4.4.11. tw_learn() and tw_learn_file()

```
tw_errno_t tw_learn(tw_t *tw, const char *cat, const char *str);

tw_errno_t tw_learn_file(tw_t *tw, const char *cat,
                         const char *path);
```

These functions analyse an input document and store its characteristics to a category's profile.

The functions take a pointer to an initialized Textweiser object (`tw_t`) as a first argument (see chapter 4.3.1 on page 16). The second parameter `cat` denotes the category to train. The third argument is the document that is an example of the category. The document can be provided as a string (*tw_learn()*) or as a path to a file (*tw_learn_file()*).

The document has to be encoded in UTF-8.

For hints on using mono-hierarchical category structures please refer to chapter 3.1.2 on page 8.

A minimum amount of documents to learn for each category is ten documents. Please take care that the documents are representative for this category and differ from each other.

The functions return an error code that indicates whether the respective function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

Both functions are thread-safe and thus can be used by more than one thread at a time.

4.4.12. tw_optimize_db()

```
tw_errno_t tw_optimize_db(tw_t *tw);
```

This function optimizes a Textweiser database with regard to performance and accuracy.

The function's argument is a pointer to an initialized Textweiser object (`tw_t`) (see chapter 4.3.1 on page 16).

The function returns an error code that indicates whether the function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

The function is thread-safe and thus can be used by more than one thread at a time.

This function should be invoked when training with a set of documents is accomplished. It has to be called whenever the structure of the system changed, for example when a category was deleted. *tw_optimize()* updates the database so that performance and accuracy increase.

4.4.13. tw_parse_config()

```
tw_errno_t tw_parse_config(const char *path, tw_config_t *config);
```

The function parses a configuration file and stores its content to a data structure. For the usage of a configuration file please refer to chapter 4.2 on page 15. Any value that is not given is set to `NULL` and `0` respectively.

The first argument is the path to the configuration file (`path`). The second argument (`config`) is a pointer to a `tw_config_t` data structure (as described in chapter 4.3.4 on page 16).

Every variable of `tw_config_t` type should be initialized using the `TW_CONFIG_INITIALIZER` macro.

The function returns an error code that indicates whether the function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

The function is thread-safe and thus can be used by more than one thread at a time.

A passed `config` variable is re-initialized on any call of the function. Any settings that may have been previously stored within will be lost.

4.4.14. tw_rename_category()

```
tw_errno_t tw_rename_category(tw_t *tw, const char *cur_name,
                              const char *new_name);
```

This function renames an existing category in a Textweiser database.

The function's first argument is a pointer to an initialized Textweiser object (`tw_t`) (see chapter 4.3.1 on page 16). The second and third argument are the current (`cur_name`) and new category name (`new_name`).

Both category names have to be encoded in UTF-8 and must not exceed a length of 255 bytes each.

For hints on using mono-hierarchical category structures please refer to chapter 3.1.2 on page 8.

The function returns an error code that indicates whether the function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

The function is thread-safe and thus can be used by more than one thread at a time.

Renaming of categories within mono-hierarchical category structures ("taxonomies") is only possible if this process does not change the relations between the categories. It has to meet the following conditions:

1. The category depth may not change
2. The relation to the direct top-level category must stay the same

Error code `TW_ECONSTR` indicates that one of these conditions is violated.

The function can only be used to rename a category and is not suitable for moving a category within a mono-hierarchical structure. If you want to move a category and change the structure, delete the category, add it at a new position and train it again.

4.4.15. tw_strerror()

```
const char * tw_strerror(tw_errno_t errnum);
```

The function takes an error indicator (`tw_errno_t`) as an argument and returns a pointer to a read-only string (`const char *`) containing the English error message.

A list of all error codes and descriptions are given in chapter 4.5.1 on page 27.

The memory of the returned string must not be freed.

The function is thread-safe and thus can be used by more than one thread at a time.

4.4.16. tw_unlearn() and tw_unlearn_file()

```
tw_errno_t tw_unlearn(tw_t *tw, const char *cat, const char *str);

tw_errno_t tw_unlearn_file(tw_t *tw, const char *cat,
                           const char *path);
```

These functions analyse an input document and undo a previously done learning operation.

The functions take a pointer to an initialized Textweiser object (`tw_t`) as a first argument (see chapter 4.3.1 on page 16). The second parameter `cat` denotes the category you trained erroneously before. The third argument is the document provided as a sting (*tw_unlearn()*) or as a file (*tw_unlearn_file()*).

The document has to be encoded in UTF-8.

For hints on using mono-hierarchical category structures please refer to chapter 3.1.2 on page 8.

The functions return an error code that indicates whether the respective function succeeded or an error occurred. For details on error handling see chapter 4.5 on page 26.

Both functions are thread-safe and thus can be used by more than one thread at a time.

After unlearning a document, the database should be updated with *tw_optimize_db()*.

4.4.17. tw_version()

```
int tw_version();
```

The function does not take an argument and returns a numeric representation of Textweiser's version.

The function is thread-safe and thus can be used by more than one thread at a time.

4.4.18. tw_version_string()

```
const char * tw_version_string();
```

The function does not take an argument and returns a pointer to a read-only string containing Textweiser's version (`const char *`), for example "1.3.0".

The memory of the returned string must not be freed.

The function is thread-safe and thus can be used by more than one thread at a time.

4.5. Error Handling

Textweiser provides an easy to use way to handle errors by evaluating the return value. Every function that may fail has an error indicator as a return value.

Any application that uses Textweiser should evaluate this error indicator to implement an adequate error handling. The return value `TW_OK` indicates that the function was successful.

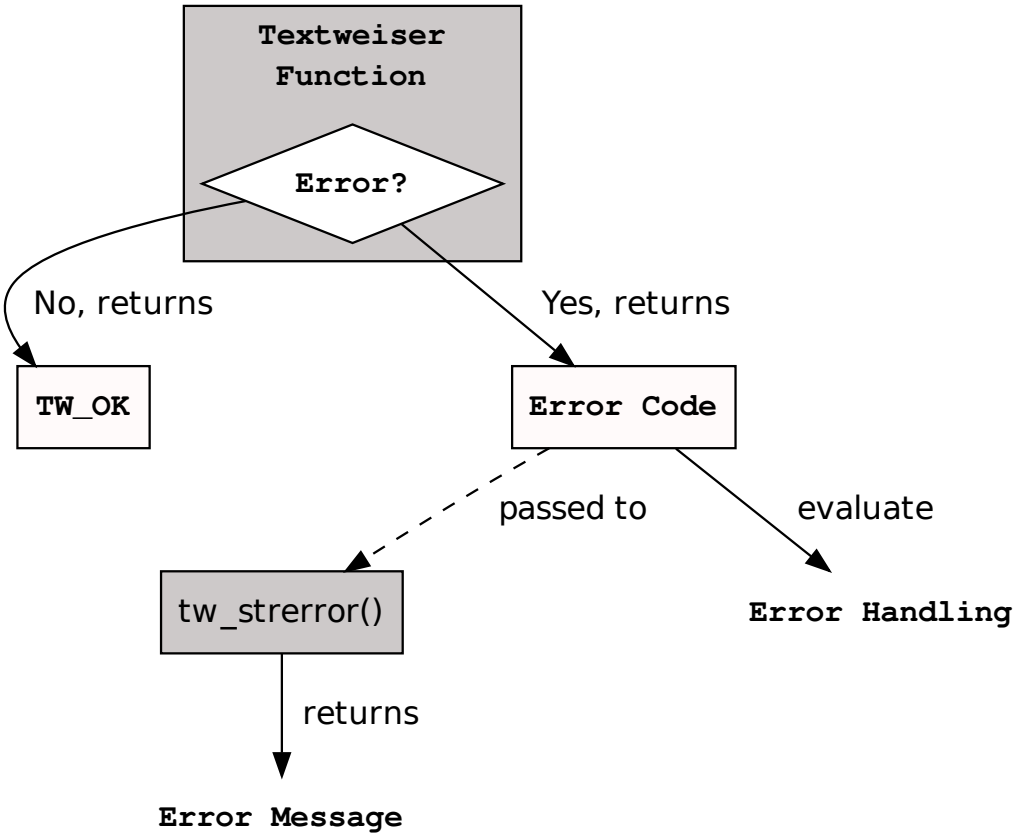


Figure 13: Flowchart of Textweiser's Error Handling

4.5.1. tw_errno_t Named Error Constants

Textweiser uses the data structure `tw_errno_t` to provide named error constants for all error cases.

Any error code may be used with *tw_strerror()* to obtain an English error message describing the error (see chapter 4.4.15, page 24).

The following table comprises all named error constants used in Textweiser version 1.3.0, accompanied by the error messages returned if passed to *tw_strerror()*.

Constant	Error Message
TW_OK	No error
TW_ENOMEM	Failed to allocate memory
TW_EARG	Invalid argument
TW_ESHORT	Insufficient input length
TW_EPREPROC	Failed to preprocess text
TW_ENOINIT	Object not initialized
TW_EIO	File input/output error
TW_EFOPEN	Failed to open file
TW_ECFG	Failed to parse configuration file
TW_ECAT	Invalid category
TW_ENOSUTF	Not a supported Unicode Transformation Format
TW_ERLOCK	Failed to lock resource
TW_ECONSTR	Constraint violated
TW_EBFMT	Invalid backup file format
TW_EBINV	Invalid backup data
TW_EDBPERM	Database denied permission
TW_EDBIO	Database input/output error
TW_EDBFULL	Database full
TW_EDBAUTH	Database authorization failed
TW_EDBCON	Failed to connect to database
TW_EDB	Internal database error
TW_EINT	Internal error

Figure 14: `tw_errno_t` Named Constants and Error Messages

4.6. Hints on Application Development

4.6.1. Determining Textweiser's Version

After including the *tw.h* header, the following preprocessor definitions are available *at compile time*.

Definition	Value
TW_VERSION_MAJOR	1
TW_VERSION_MINOR	3
TW_VERSION_BUGFIX	0
TW_VERSION_STRING	"1.3.0"

Figure 15: Version Information at Compile Time

To determine Textweiser's version at runtime, use *tw_version()* or *tw_version_string()* (see chapter 4.4.18, page 25).

5. Commandline Interface

Textweiser includes four applications that allow to use all essential functionality on the commandline. These applications can also be utilized in scripts, for example to automate common administration tasks such as optimizing the database periodically.

Every Textweiser application provides a short help on its usage if invoked with the `-h` parameter: *tw-admin(1)*, *tw-learn(1)*, *tw-classify(1)* and *tw-backup(1)*.

The first section introduces the parameters required to establish a connection to the database. A detailed overview and usage examples of the applications are given afterwards.

5.1. Connecting to the Database

Every application included in the Textweiser software distribution needs to access the database. The required connection settings may either be passed to the application directly on the commandline or be stored in a configuration file.

All applications accept the following commandline parameters:

Short-	Long-Option	Parameter	Type	Example
-d	--db_name	Database name	String	`tw-db`
-s	--host	Name of the database server	String	`localhost`
-u	--user	Username	String	`doe`
-w	--passwd	Password	String	`secret`
-p	--port	Port of the database server	Number	`1433`
-t	--instance	SQL Server instance	String	`SQLEXPRESS`
-e	--encrypt	-	-	-
	--trust-cert	-	-	-

Figure 16: Parameters used to connect to the Database

The `--instance` option is only available in the SQL Server version of Textweiser and allows to specify the SQL Server instance that should be used.

The option `--encrypt` enables encryption of the communication to the database, if the database supports encryption. If no encrypted connection can be established, the application will abort with an appropriate error message. If you want to trust the server's certificate without validation, pass the `--trust-cert` option, which is required in order to use self-signed certificates.

Whenever no port is specified, the application will use the default port of the database software.

If no password is given as a parameter, the user can enter the password interactively. For security reasons, the entered password will not be echoed on the commandline.

All settings may be stored in a configuration file as well. The expected configuration entries consist of simple key/value pairs (see chapter 4.2 on page 15).

A configuration file can be selected by passing either the `-f` or `--config` option followed by the path of the file within the file system. If other connection parameters are giving directly on the commandline, these override those that may have been set by a configuration file.

If SQLite is used as database software, the parameter of "`-d`" or "`--db_name`" denotes the path to the database within the file system. Besides that, no other database connection options are required or available.

5.2. Common Options

Besides the options used to specify how to connect to the database, all Textweiser applications provide the following set of common options:

Short-	Long-Option	Description
`-v`	`--verbose`	Enable verbose output
`-V`	`--version`	Show version information
`-h`	`--help`	Show short help

Figure 17: Common Options

5.3. tw-admin: Textweiser Administration

tw-admin provides the possibility to create and administrate Textweiser databases on the commandline. For example, new categories can be added, existing categories deleted or renamed.

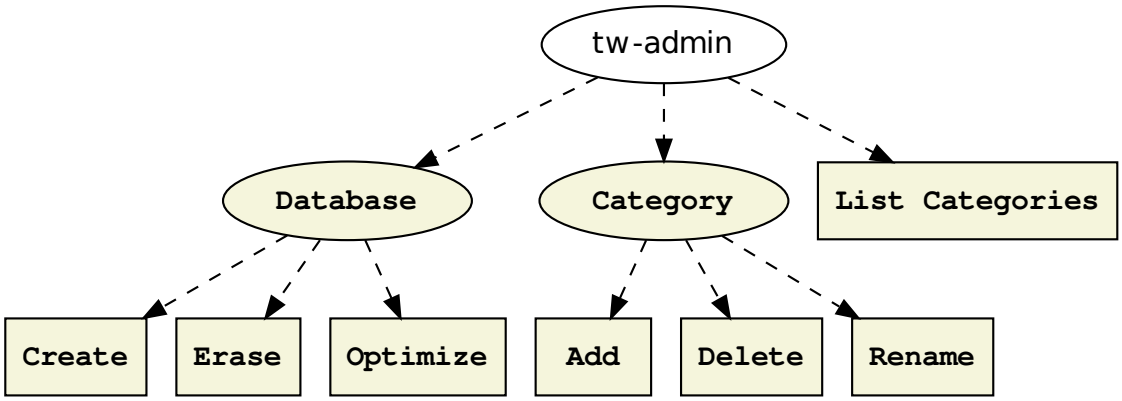


Figure 18: tw-admin: Textweiser Administration

To use a specific functionality provided by `tw-admin`, the corresponding mode has to be activated by passing an option:

Operations on categories require the name of the respective category. It has to be given as an argument to the `-c` or `--cat` option in order to add, delete or rename a category. In the latter case, the new category name is expected to be given as an argument to the `-n` or `--cat_new` option. All category names have to be UTF-8 encoded and are restricted to a maximum length of 255 bytes.

tw-admin handles both flat and mono-hierarchical category structures ("taxonomies"). If taxonomies are used, any renaming operation is subject to the restrictions that the new category name has to be of the same category depth and keep the same direct top-level category. For further details on how to use hierarchies have a look at chapter 3.1 on page 8.

The available parameters used to connect to the database are described in chapter 5.1 on page 29.

Short-	Long-Option	Description of Mode
-C	--create	Create a new database
-A	-add-cat	Add a new category
-D	--del-cat	Delete an existing category
-R	--ren-cat	Rename an existing category
-L	--list	List all categories
-O	--optimize	Optimize all data records
-E	--erase	Erase all data records

Figure 19: tw-admin: Options and Modes

5.3.1. Usage Example

The following examples assume the SQLite version of Textweiser is used and utilize verbose processing mode.

First, a new database is created and a few categories are added, one containing a typing error. A category listing is requested afterwards.

```
$ tw-admin -v -d textweiser.sqlt -C
Creating Textweiser tables in textweiser.sqlt
$ tw-admin -v -d textweiser.sqlt -A -c Sales
Adding category "Sales"
$ tw-admin -v -d textweiser.sqlt -A -c Projcets
Adding category "Projcets"
$ tw-admin -v -d textweiser.sqlt -L
Categories in textweiser.sqlt:
01: Projcets
02: Sales
```

The typing error in the category name "Projcets" will now be fixed by renaming the category.

```
$ tw-admin -v -d textweiser.sqlt -R -c Projcets -n Projects
Renaming category "Projcets" to "Projects"
$ tw-admin -v -d textweiser.sqlt -L
Categories in textweiser.sqlt:
01: Projects
02: Sales
```

5.4. tw-learn: Learn Category Characteristics

tw-learn determines category characteristics using a set of representative documents. Similar documents can then be automatically classified. If a document has been learned erroneously as an example of a category, *tw-learn* is able to unlearn characteristics by updating the learned associations and optimizing the database afterwards.

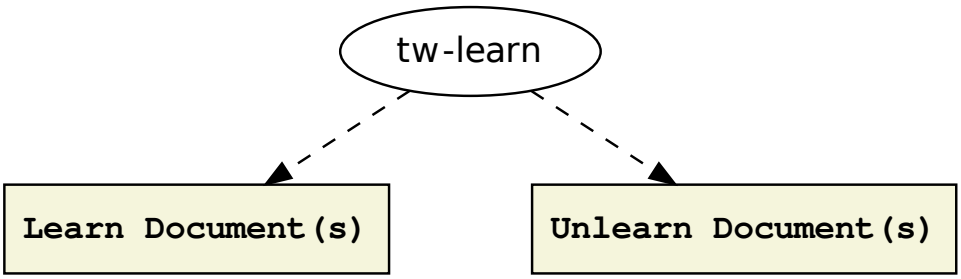


Figure 20: tw-learn: Learning of Category Characteristics

In order to instruct *tw-learn* to determine and learn characteristics, pass the paths to the representative documents. The category they belong to is specified using the `-c` or `--cat` option. If it is necessary to unlearn a document's characteristics and the resulting associations, the option `-U` or `--unlearn` switches *tw-learn* to its unlearning mode.

The available parameters used to connect to the database are described in chapter 5.1 on page 29.

5.4.1. Usage Example

The following examples assume the SQLite version of Textweiser is used and utilize verbose processing mode.

First, *tw-learn* is used to determine and learn the characteristics of the documents per category and associate these with the respective category.

```
$ tw-learn -v -d textweiser.sqlt -c Sales sales_1.txt sales_2.txt
# Processing sales_1.txt... OK
# Processing sales_2.txt... OK
Learned 2 documents of category "Sales"
$ tw-learn -v -d textweiser.sqlt -c Projects projects_1.txt \
    projects_2.txt
# Processing projects_1.txt... OK
# Processing projects_2.txt... OK
Learned 2 documents of category "Projects"
```

In order to give an example on unlearning, a document will be learned as an example of the wrong category. The learning process will then be reverted and the document assigned to the correct category.

After unlearning a document, the database will automatically be optimized to update all data records accordingly. In contrast to using the library directly, this operation does not have to be executed manually.

```
$ tw-learn -v -d textweiser.sqlt -c Sales projects_3.txt
# Processing projects_3.txt... OK
Learned 1 document of category "Sales"
$ tw-learn -v -d textweiser.sqlt -c Sales -U projects_3.txt
# Processing projects_3.txt... OK
Optimizing database
Unlearned 1 document of category "Sales"
$ tw-learn -v -d textweiser.sqlt -c Projects projects_3.txt
# Processing projects_3.txt... OK
Learned 1 document of category "Projects"
```

5.5. tw-classify: Classify Unknown Documents

Unknown documents can automatically be classified using *tw-classify* as soon as the Textweiser database has been initialized with a set of categories and trained using representative documents. During classification the unknown documents are analysed and their determined characteristics are compared to those of the trained categories. By default, *tw-classify* uses a single thread and prints only the most likely category for each document.

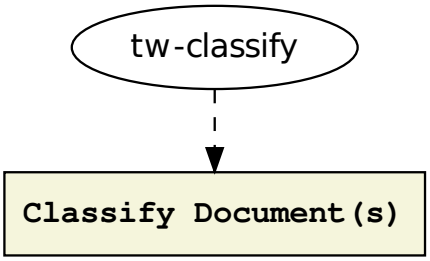


Figure 21: tw-classify: Classifying Unknown Documents

tw-classify requires a set of paths to unknown documents as arguments only.

The number of threads to use for classification may optionally be set using the `-x` or `--threads` option. Increasing the number of threads may lead to increased processing speed, especially on multicore systems.

The `-n` or `--show` option allows to specify the number of result categories to be shown along with their determined probabilities.

The `-r` or `--distribute` option allows to distribute the determined probabilities (see chapter 3.2 on page 9 for an explanation).

Whenever a set of classification results shares equal probabilities, parent categories will precede their subcategories. The option `-b` or `--sub-first` changes this sorting behaviour and places subcategories in front of their parent categories.

Short-	Long-Option	Parameter	Description
`-x`	`--threads`	Number	Use the given *number* of threads
`-n`	`--show`	Number	Show at max *number* results
`-r`	`--distribute`	--	Use distributed probabilities
`-b`	`--sub-first`	--	On equal probabilities, subcategories first

Figure 22: tw-classify: Classification Options

The available parameters used to connect to the database are described in chapter 5.1 on page 29.

5.5.1. Usage Example

The following examples assume the SQLite version of Textweiser is used.

The following examples show how *tw-classify* classifies four documents using two threads – once using the default output settings and once using verbose processing mode combined with a user-defined setting regarding the amount of results to show.

```
$ tw-classify -d textweiser.sqlt -x 2 text_1.txt text_2.txt \
    text_3.txt text_4.txt
text_1.txt: Sales
text_2.txt: Sales
text_3.txt: Projects
text_4.txt: Projects
$ tw-classify -v -d textweiser.sqlt -x 2 -n 5 text_1.txt \
    text_2.txt text_3.txt text_4.txt
Classification results for text_1.txt:
  01:            Sales -> 100.00%
  02:         Projects -> 41.25%

Classification results for text_2.txt:
  01:            Sales -> 100.00%

Classification results for text_3.txt:
  01:         Projects -> 100.00%
  02:            Sales -> 16.38%

Classification results for text_4.txt:
  01:         Projects -> 100.00%
```

5.6. tw-backup: Backup and Restore the Database

tw-backup is used to create and restore Textweiser database backups. When restoring from a backup, all possibly existing data records of the selected Textweiser database will be erased and replaced by those of the backup.

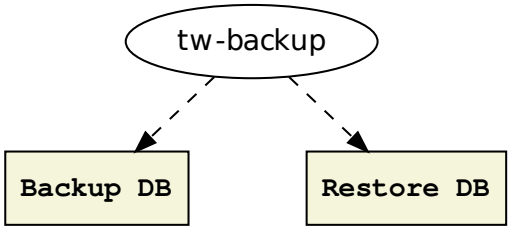


Figure 23: tw-backup: Textweiser Backup

The modes can be activated using the options `-B` or `--backup` and `-R` or `--restore` respectively. It is mandatory to specify a backup file as well: `-o` or `--output` is used to set the output file in backup mode while `-i` or `--input` expects a path to a previously created backup file as an argument.

The available parameters used to connect to the database are described in chapter 5.1 on page 29.

5.6.1. Usage Example

The following examples assume the SQLite version of Textweiser is used and utilize verbose processing mode.

All categories known to the current database are displayed and a backup is created afterwards.

```
$ tw-admin -v -d example.sqlt -L
Categories in example.sqlt:
01: Projects
02: Sales
$ tw-backup -v -d example.sqlt -B -o example.bup
Storing backup of example.sqlt to example.bup
```

A new database is created and initialized using the backup file created before.

```
$ tw-admin -v -d restored.sqlt -C
Creating Textweiser tables in restored.sqlt
$ tw-backup -v -d restored.sqlt -R -i example.bup
Restoring backup from example.bup to restored.sqlt
$ tw-admin -v -d restored.sqlt -L
Categories in restored.sqlt:
01: Projects
02: Sales
```

A. Example Application: add-learn.c

```c
#include <stdio.h>
#include <stdlib.h>

#include <tw.h>


struct cat
{
    const char *name;
    const char *text;
};

struct cat cats[] =
{
    { "Cinema",  "Several new films start this weekend." },
    { "Weather", "Today it is a bit cloudy." }
};


int main(int argc, char *argv[])
{
    tw_errno_t  rv  = TW_OK;
    tw_config_t cfg = TW_CONFIG_INITIALIZER;
    tw_t        tw  = TW_INITIALIZER;
    short       i   = 0;

    /* Initialize a Textweiser object using the SQLite
     * database backend. */
    cfg.db_name = "example.sqlt";

    rv = tw_init(&tw, &cfg);

    if (rv != TW_OK)
    {
        tw_free(&tw);

        fprintf(stderr, "Failed to initialize: %s\n",
                tw_strerror(rv));

        return EXIT_FAILURE;
    }


    for (i = 0; i < (sizeof(cats) / sizeof(struct cat)); i++)
    {
        printf("Adding category: \"%s\"\n", cats[i].name);

        rv = tw_add_category(&tw, cats[i].name);

        if (rv != TW_OK)
        {
```

```
            tw_free(&tw);

            fprintf(stderr, "Failed to add category: %s\n",
                    tw_strerror(rv));

            return EXIT_FAILURE;
        }


        printf("  Learning text: \"%s\"\n", cats[i].text);

        rv = tw_learn(&tw, cats[i].name, cats[i].text);

        if (rv != TW_OK)
        {
            tw_free(&tw);

            fprintf(stderr, "Failed to learn text: %s\n",
                    tw_strerror(rv));

            return EXIT_FAILURE;
        }
    }


    tw_free(&tw);


    return EXIT_SUCCESS;
}
```

The following output shows an example execution of the application:

```
Adding category: "Cinema"
  Learning text "Several new films start this weekend."
Adding category: "Weather"
  Learning text: "Today it is a bit cloudy."
```

B. Example Application: classify.c

```c
#include <stdio.h>
#include <stdlib.h>

#include <tw.h>

int main(int argc, char *argv[])
{
    tw_errno_t   rv     = TW_OK;
    tw_config_t  cfg    = TW_CONFIG_INITIALIZER;
    tw_prob_t  **probs  = NULL;
    const char  *string = "The house prices have risen.";
    tw_t         tw     = TW_INITIALIZER;

    /* Initialize a Textweiser object using the SQLite
     * database backend. */
    cfg.db_name = "example.sqlt";

    rv = tw_init(&tw, &cfg);

    if (rv != TW_OK)
    {
        fprintf(stderr, "Failed to initialize: %s\n",
                tw_strerror(rv));

        return EXIT_FAILURE;
    }


    rv = tw_classify(&tw, string, 2, &probs);

    tw_free(&tw);

    if (rv == TW_OK)
    {
        if (probs)
        {
            short i = 0;

            for (i = 0; probs[i]; i++)
            {
                printf("Category \"%s\" -> %.2f%%\n",
                       probs[i]->category, probs[i]->probability);
            }

            tw_free_prob_t(probs);
        }
        else
        {
            puts("No results");
        }
```

```
        return EXIT_SUCCESS;
    }
    else
    {
        fprintf(stderr, "Failed to classify: %s\n",
                tw_strerror(rv));

        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

The following output shows an example execution of the application:

```
Category "Economy & Markets" -> 100.00%
Category "Holidays" -> 13.02%
```

C. References

→ Lingua-Systems' Textweiser SDK product website,
http://www.lingua-systems.com/text-classification/

→ Textweiser SDK software specification for version 1.3.0

→ The Unicode Standard,
http://www.unicode.org/

→ RFC 2279: "UTF-8, a transformation format of ISO 10646",
http://www.ietf.org/rfc/rfc2279.txt

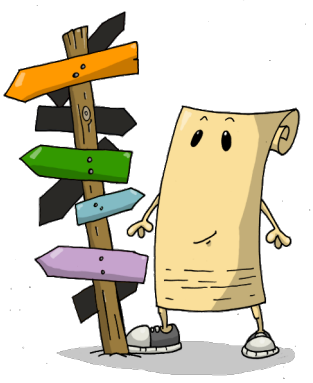
→ SQLite,
http://www.sqlite.org/

→ Microsoft SQL Server,
http://www.microsoft.com/sqlserver/

→ MSDN: "Installing SQL Server Native Client",
http://msdn.microsoft.com/en-us/library/ms131321.aspx

→ MSDN: "Encrypting Connections to SQL Server",
http://msdn.microsoft.com/en-us/library/ms189067.aspx

→ MSDN: "Using Encryption Without Validation",
http://msdn.microsoft.com/en-us/library/ms131691.aspx



http://www.lingua-systems.com/text-classification/

Index

A

application programming interface (API) 12
applications . . . . . *see* commandline applications

B

backup 19, 35

C

category structure . . . . . *see* hierarchy
certificates . . . . . *see* encryption
classification options . . . . . *see* tw_classify_opt_t
classification result . . . . . *see* tw_prob_t
commandline applications 29
    tw-admin 30
    tw-backup 35
    tw-classify 33
    tw-learn 31
configuration file 15

D

data structures 16
    tw_classify_opt_t 16
    tw_config_t 17
    tw_errno_t . . . . . *see* tw_errno_t
    tw_prob_t 16
    tw_t 16
database connection 10, 15, 17, 22, 29
deinstalling the software 7
dependencies . . . . . *see* requirements

E

encryption 10, 15, 17, 29
    certificate 11, 29
    Microsoft SQL Server 11
    self-signed certificate 11, 29
    self-signed certificates 18
    SSL 11
error codes . . . . . *see* tw_errno_t
error handling 26
    named constants 27
example application
    add-learn.c 36
    classify.c 38

F

flat category structures 8
functions
    administration 12
    auxiliaries 14
    classification 14
    learning 13
    resource handling 13

H

hierarchy 8
    mono-hierarchical 8
    notation 9
    specifics 9

I

installing the software 7

M

Microsoft SQL Server 7, 11
mono-hierarchical category structure 8

N

named error constants 27

P

probability
    absolute 9
    calculation 9
    distributed 9

R

requirements 7
restore 19, 35

S

self-signed certificates . . . . . *see* encryption

T

taxonomy 8
training 8, 22, 31
tw-admin 30
tw-backup 35
tw-classify 33
tw-learn 31
tw_add_category() 19
tw_backup_db() 19
tw_classify() 20
TW_CLASSIFY_DEFAULT 16
tw_classify_file() 20
tw_classify_file_v2() 16, 20
tw_classify_opt_t 16, 20
TW_CLASSIFY_PABS 16
TW_CLASSIFY_PDIST 16
TW_CLASSIFY_SPARF 16, 17
TW_CLASSIFY_SSUBF 16, 17
tw_classify_v2() 16, 20
TW_CONFIG_INITIALIZER 17, 23
tw_config_t 17, 21
tw_create_db() 20
tw_delete_category() 19
TW_ENCRYPT_OFF 17

TW_ENCRYPT_ON 11, 17
TW_ENCRYPT_TRUST_CERT 11, 17
tw_erase_db() 20
tw_errno_t 27
TW_EARG 27
TW_EBFMT 27
TW_EBINV 27
TW_ECAT 27
TW_ECFG 27
TW_ECONSTR 27
TW_EDB 27
TW_EDBAUTH 27
TW_EDBCON 27
TW_EDBFULL 27
TW_EDBIO 27
TW_EDBPERM 27
TW_EFOPEN 27
TW_EINT 27
TW_EIO 27
TW_ENOINIT 27
TW_ENOMEM 27
TW_ENOSUTF 27
TW_EPREPROC 27
TW_ERLOCK 27
TW_ESHORT 27
TW_OK 27
tw_free() 21
tw_free_categories() 21
tw_free_config_t() 21
tw_free_prob_t() 21
tw_get_categories() 22
tw_init() 22
TW_INITIALIZER 16, 22
tw_learn(), tw_learn_file() 22
tw_optimize_db() 23
tw_parse_config() 23
tw_prob_t 16, 20, 21
tw_rename_category() 23
tw_restore_db() 19
tw_strerror() 24
tw_t 16, 21, 22
tw_unlearn(), tw_unlearn_file() 24
tw_version() 25, 28
TW_VERSION_BUGFIX 28
TW_VERSION_MAJOR 28
TW_VERSION_MINOR 28
TW_VERSION_STRING 28
tw_version_string() 25, 28